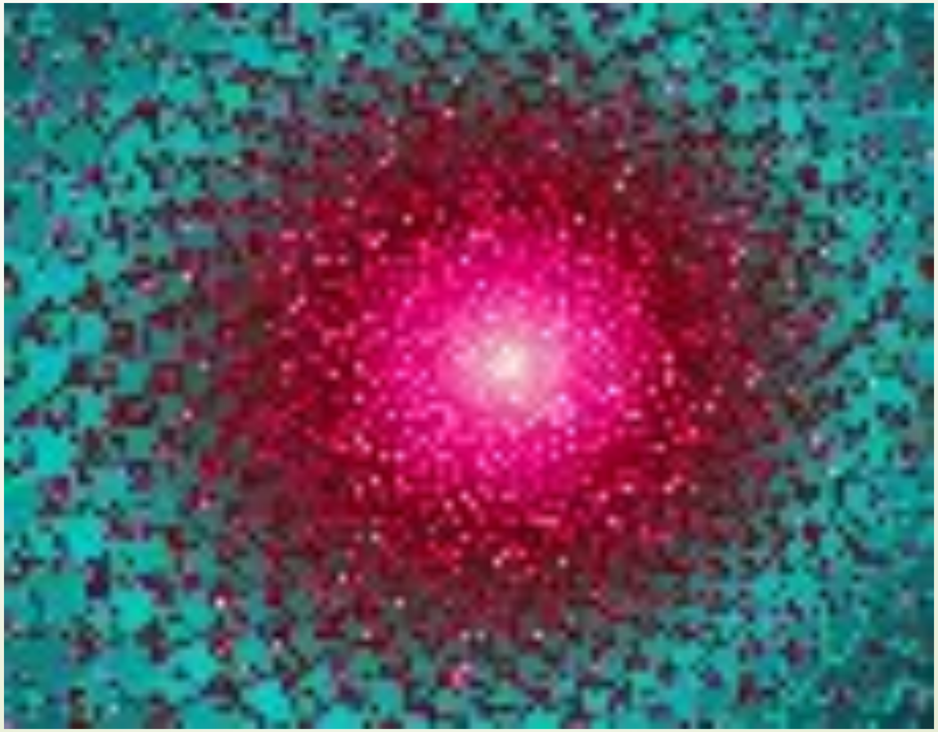
Recursion



Ex. 1: The Handshake Problem

There are n people in a room. If each person shakes hands once with every other person. What is the total number h(n) of handshakes?

h(n) = h(n-1) + n-1 **h(4) = h(3) + 3** **h(3) = h(2) + 2** **h(2) = 1**

h(n): Sum of integer from 1 to n-1 = n(n-1) / 2

Recursion

- In some problems, it may be natural to define the problem in terms of the problem itself.
- Recursion is useful for problems that can be represented by a simpler version of the same problem.
- Example: the factorial function

6! = 6 * 5 * 4 * 3 * 2 * 1

We could write:

6! = 6 * 5!

- In programming: A **recursive procedure** is a procedure which calls itself.

  Caution: The recursive procedure call must use a different argument that the original one: otherwise the procedure would always get into an infinite loop…

Recursive Function

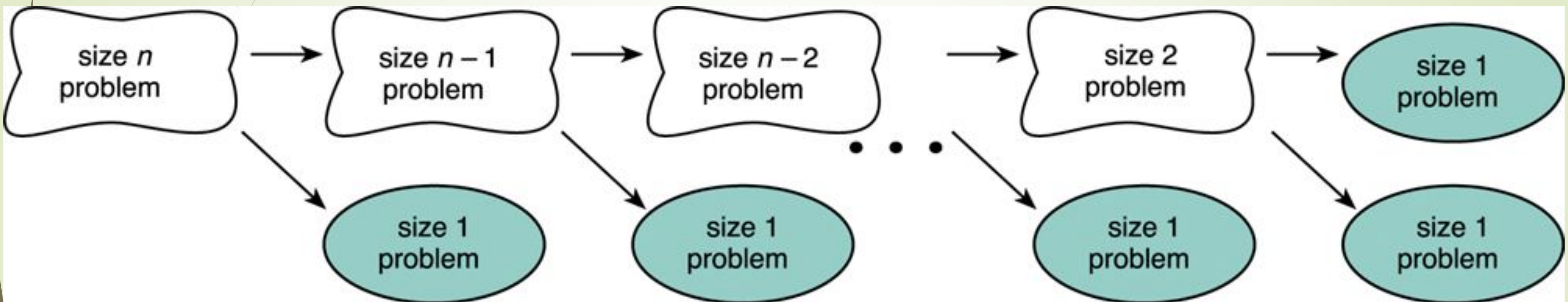
- A function that replicates itself again and again until the base case is not achieved.

- **Base case(s).**
  - Values of the input variables for which we perform no recursive calls are called **base cases** (there should be at least one base case).
  - Every possible chain of recursive calls **must** eventually reach a base case.
- ***Recursive calls.***
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

- *Recursive Function:–* a function that calls itself
    - Directly or indirectly
- Each recursive call is made with a new, independent set of arguments
    - Previous calls are suspended
- Allows very simple programs for very complex problems

- The problem can be reduced entirely to simple cases by calling the recursive function.
  - *If this is a simple case*
    solve it
    else
    redefine the problem using recursion

Splitting a Problem into Smaller Problems



- Assume that the problem of size 1 can be solved easily (i.e., the simple case).
- We can recursively split the problem into a problem of size 1 and another problem of size n-1.

Example 2: factorial function
// Linear Recursion

In general, we can express the factorial function as follows:

```
n! = n * (n-1)!
```

Is this correct? Well… almost.

The factorial function is only defined for *positive* integers. So we should be a bit more precise:

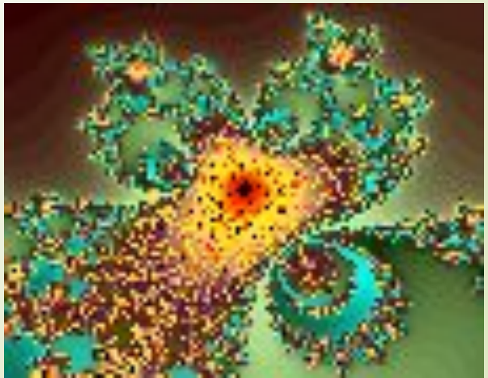
```
n! = 1     (if n is equal to 1)
n! = n * (n-1)!    (if n is larger than 1)
```

factorial function

The C++ equivalent of this definition:

```
int fac(int numb){
    if(numb<=1)
        return 1;
    else
        return numb * fac(numb-1);
}
```

***recursion* means that a function calls itself**



factorial function

- Assume the number typed is 3, that is, numb=3.

```
fac(3) :
 3 <= 1 ?          No.
 fac(3) = 3 * fac(2)
    fac(2) :
       2 <= 1 ?     No.
       fac(2) = 2 * fac(1)

           fac(1) :
              1 <= 1 ?    Yes.
              return 1

        fac(2) = 2 * 1 = 2
        return fac(2)

  fac(3) = 3 * 2 = 6
  return fac(3)

  fac(3)  has the value 6
```

```
int fac(int numb){
   if(numb<=1)
      return 1;
   else
      return numb * fac(numb-1);
}
```

- factorial 3
- =
- 3 * factorial 2
- =
- 3 * (2 * factorial 1)
- =
- 3 * (2 * (1 * factorial 0))
- =
- 3 * (2 * (1 * 1))
- =
- 3 * (2 * 1)
- =
- 3 * 2
- =
- 6

Factorial function

For certain problems (such as the factorial function), a recursive solution often leads to short and elegant code. Compare the recursive solution with the iterative solution:

Recursive solution

```
int fac(int numb){
  if(numb<=1)
    return 1;
  else
   return numb*fac(numb-1);
}
```

Iterative solution

```
int fac(int numb){
    int product=1;
    while(numb>1){
   product *= numb;
   numb--;
    }
    return product;
 }
```

Recursion

If we use iteration, we must be careful not to create an infinite loop by accident:

```
for(int incr=1; incr!=10;incr+=2)
   ...



int result = 1;
while(result >0){
   ...
   result++;
}
```

Oops!

Oops!

Recursion

Similarly, if we use recursion we must be careful not to create an infinite chain of function calls:

```
int fac(int numb){
   return numb * fac(numb-1);
}
```

Or:

```
int fac(int numb){
   if (numb<=1)
      return 1;
   else
      return numb * fac(numb+1);
}
```

Oops!

No termination condition

Oops!

Recursion

We must always make sure that the recursion *bottoms out*:

- A recursive function must contain at least one non-recursive branch.
- The recursive calls must eventually lead to a non-recursive branch.

Recursion

- Recursion is one way to decompose a task into smaller subtasks. At least one of the subtasks is a smaller example of the same task.
- The smallest example of the same task has a non-recursive solution.

Example: The factorial function

```
n! = n * (n-1)! and 1! = 1
```

Direct Computation Method

- Fibonacci numbers:

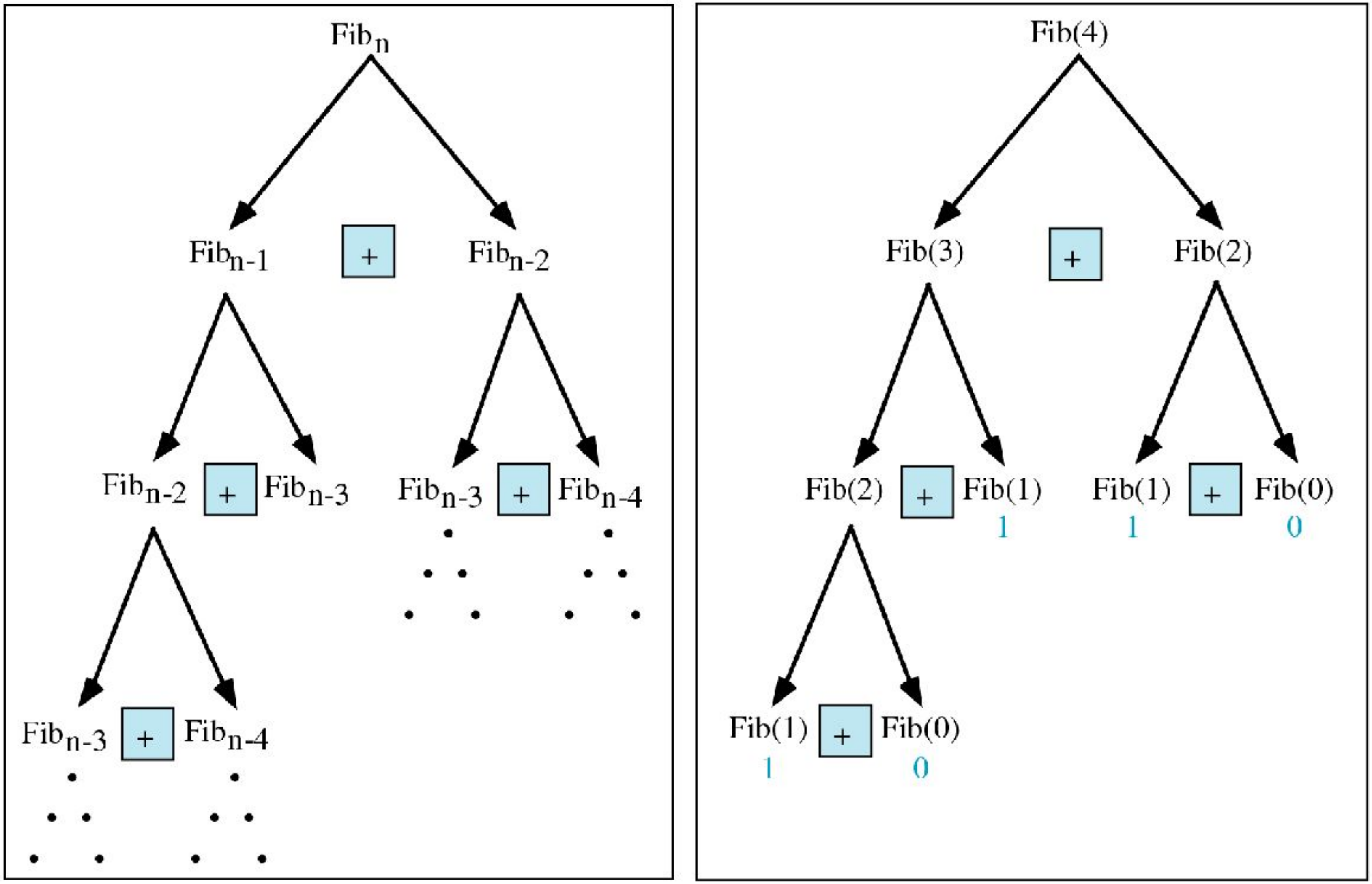
  `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`

  where each number is the sum of the preceding two.

- Recursive definition:
  - `F(0) = 0;`
  - `F(1) = 1;`

Example 3: Fibonacci numbers

```
//Calculate Fibonacci numbers using recursive function.
//A very inefficient way, but illustrates recursion well
// Binary Recursion

int fib(int number)
{
    if (number == 0) return 0;
    if (number == 1) return 1;
    return (fib(number-1) + fib(number-2));
}

int main(){     // driver function
    int inp_number;
    cout << "Please enter an integer: ";
    cin >> inp_number;
    cout << "The Fibonacci number for "<< inp_number
      << " is "<< fib(inp_number)<<endl;
  return 0;
}
```


Fib n
Fib n-1
+
Fib n-2
Fib n-2
+
Fib n-3
Fib n-3
+
Fib n-4
Fib n-3
+
Fib n-4
Fib(4)
Fib(3)
+
Fib(2)
Fib(2)
+
Fib(1)
1
Fib(1)
1
+
Fib(0)
0
Fib(1)
1
+
Fib(0)
0

(a) Fib(n)

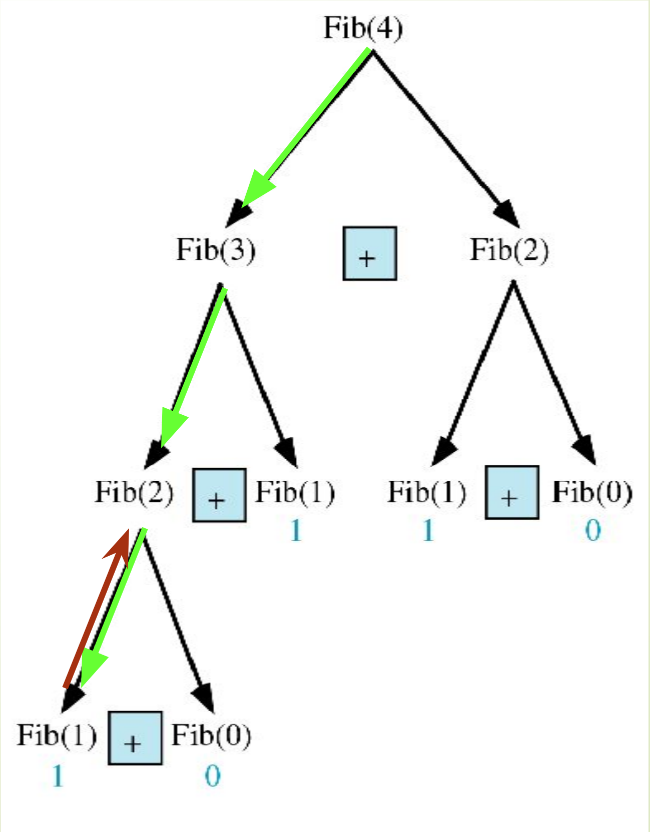
(b) Fib(4)

Trace a Fibonacci Number

- Assume the input number is 4, that is, num=4:

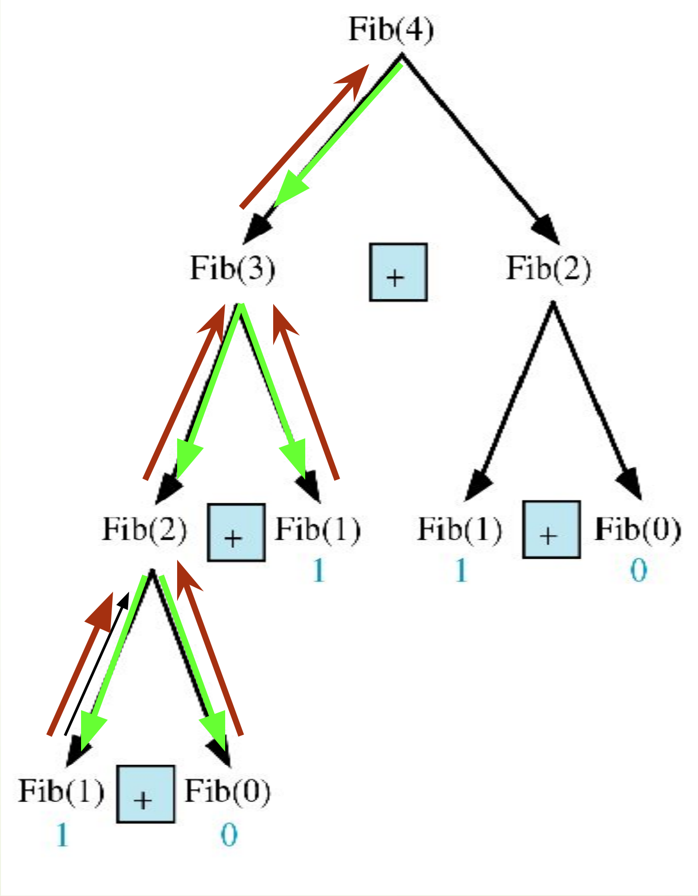
```
fib(4):
    4 == 0 ? No;    4 == 1?  No.
    fib(4) = fib(3) + fib(2)
    fib(3):
        3 == 0 ? No; 3 == 1? No.
        fib(3) = fib(2) + fib(1)
        fib(2):
            2 == 0? No; 2==1? No.
            fib(2) = fib(1)+fib(0)
           fib(1):
              1== 0 ? No; 1 == 1? Yes.
            fib(1) = 1;
            return fib(1);
```

```
int fib(int num)
{
    if (num == 0) return 0;
    if (num == 1) return 1;
    return
      (fib(num-1)+fib(num-2));
}
```



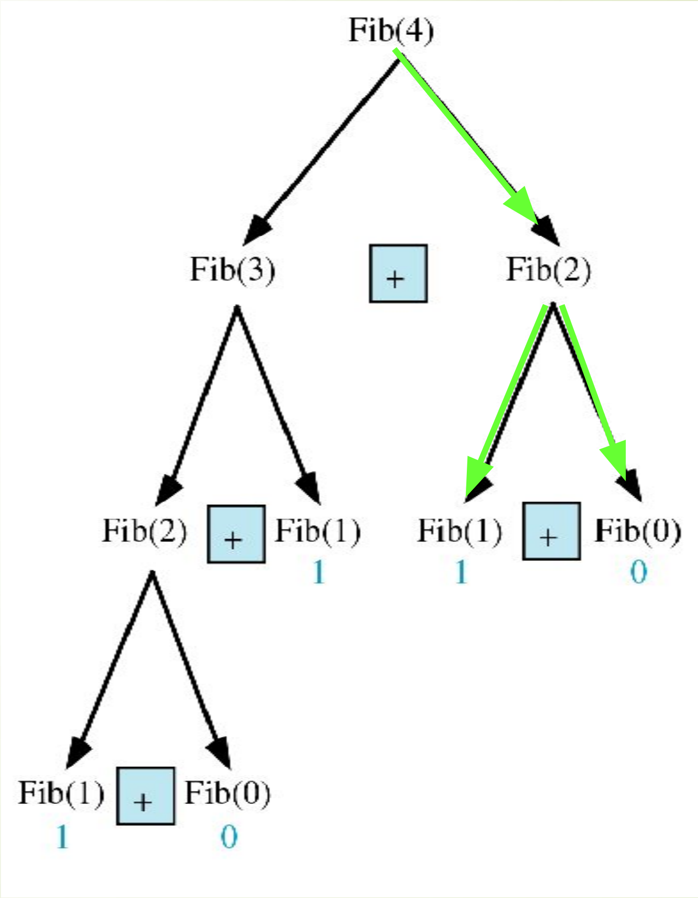
Trace a Fibonacci Number

```
fib(0):
    0 == 0 ?  Yes.
 fib(0) = 0;
    return fib(0);
fib(2) = 1 + 0 = 1;
return fib(2);
fib(3) = 1 + fib(1)
 fib(1):
    1 == 0 ? No; 1 == 1? Yes
    fib(1) = 1;
     return fib(1);
fib(3) = 1 + 1 = 2;
return fib(3)
```



Trace a Fibonacci Number

```
fib(2):
     2 == 0 ? No; 2 == 1?    No.
     fib(2) = fib(1) + fib(0)
     fib(1):
       1== 0 ? No; 1 == 1?  Yes.
     fib(1) = 1;
     return fib(1);
     fib(0):
        0 == 0 ?    Yes.
     fib(0) = 0;
         return fib(0);
     fib(2) = 1 + 0 = 1;
     return fib(2);
 fib(4) = fib(3) + fib(2)
        = 2 + 1 = 3;
 return fib(4);
```



Example 4: Fibonacci number w/o recursion

```
//Calculate Fibonacci numbers iteratively
//much more efficient than recursive solution

int fib(int n)
{
   int f[100];
   f[0] = 0; f[1] = 1;
   for (int i=2; i<= n; i++)
        f[i] = f[i-1] + f[i-2];
   return f[n];
}
```

Fibonacci Numbers

- Fibonacci numbers can also be represented by the following formula.

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

Example 5: Binary Search

- Search for an element in an array
  - Binary search
- Binary search
  - Compare the search element with the middle element of the array
  - If not equal, then apply binary search to half of the array (if not empty) where the search element would be.

Binary Search with Recursion

```
// Searches an ordered array of integers using recursion
int bsearchr(const int data[], // input: array
             int first,         // input: lower bound
             int last,         // input: upper bound
             int value          // input: value to find
         )// output: index if found,  otherwise return -1

{   //cout << "bsearch(data, "<<first<< ", last "<< ", "<<value << "); "<<endl;
    int middle = (first + last) / 2;
    if (data[middle] == value)
        return middle;
    else if (first >= last)
        return -1;
    else if (value < data[middle])
        return bsearchr(data, first, middle-1, value);
    else
        return bsearchr(data, middle+1, last, value);
}
```

Binary Search

```
int main() {
     const int array_size = 8;
     int list[array_size]={1, 2, 3, 5, 7, 10, 14, 17};
     int search_value;

     cout << "Enter search value: ";
     cin >> search_value;
     cout << bsearchr(list,0,array_size-1,search_value)
          << endl;

     return 0;
}
```

Recursion General Form

- How to write recursively?

```
int recur_fn(parameters){
  if(stopping condition)
   return stopping value;
  // other stopping conditions if needed
  return function of recur_fn(revised parameters)

}
```

Example 6: exponential func

- How to write `exp(int numb, int power)` recursively?

```
int exp(int numb, int power){
   if(power ==0)
      return 1;
   return numb * exp(numb, power -1);
}
```

Binary Search w/o recursion

```
// Searches an ordered array of integers
int bsearch(const int data[], // input: array
            int size,     // input: array size
            int value     // input: value to find
           ){             // output: if found,return
                          // index; otherwise, return -1

    int first, last, upper;
        first = 0;
        last = size - 1;
    while (true) {
            middle = (first + last) / 2;
            if (data[middle] == value)
                return middle;
            else if (first >= last)
                return -1;
            else if (value < data[middle])
         last = middle - 1;
            else
         first = middle + 1;
      }
}
```

Example 7: Towers of Hanoi

- Only one disc could be moved at a time
- A larger disc must never be stacked above a smaller one
- One and only one extra needle could be used for intermediate storage of discs

Towers of Hanoi

```
void hanoi(int from, int to, int num)
{
    int temp = 6 - from - to; //find the temporary
                              //storage column
    if (num == 1){
        cout << "move disc 1 from " << from
             << " to " << to << endl;
    }
    else {
        hanoi(from, temp, num - 1);
        cout << "move disc " << num << " from " << from
             << " to " << to << endl;
        hanoi(temp, to, num - 1);
    }
}
```

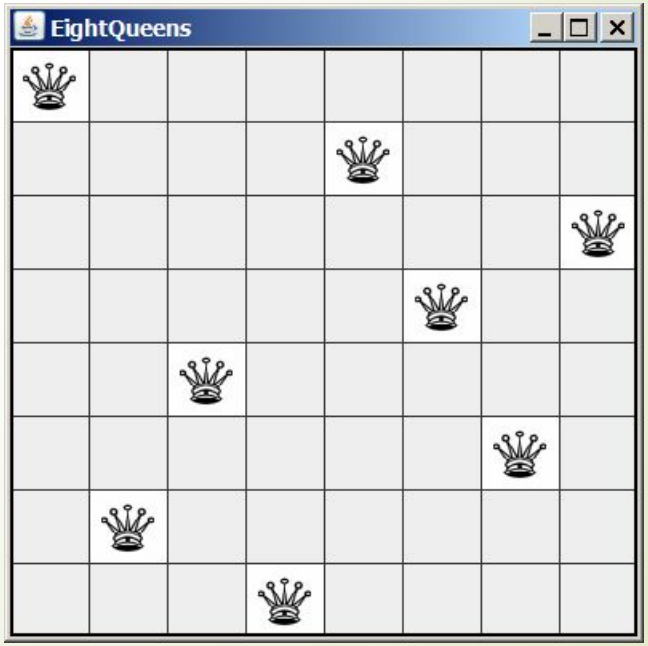
Towers of Hanoi

```
int main() {
    int num_disc;  //number of discs

    cout << "Please enter a positive number (0 to quit)";
    cin >> num_disc;

    while (num_disc > 0){
        hanoi(1, 3, num_disc);
        cout << "Please enter a positive number ";
        cin >> num_disc;
    }
    return 0;
}
```

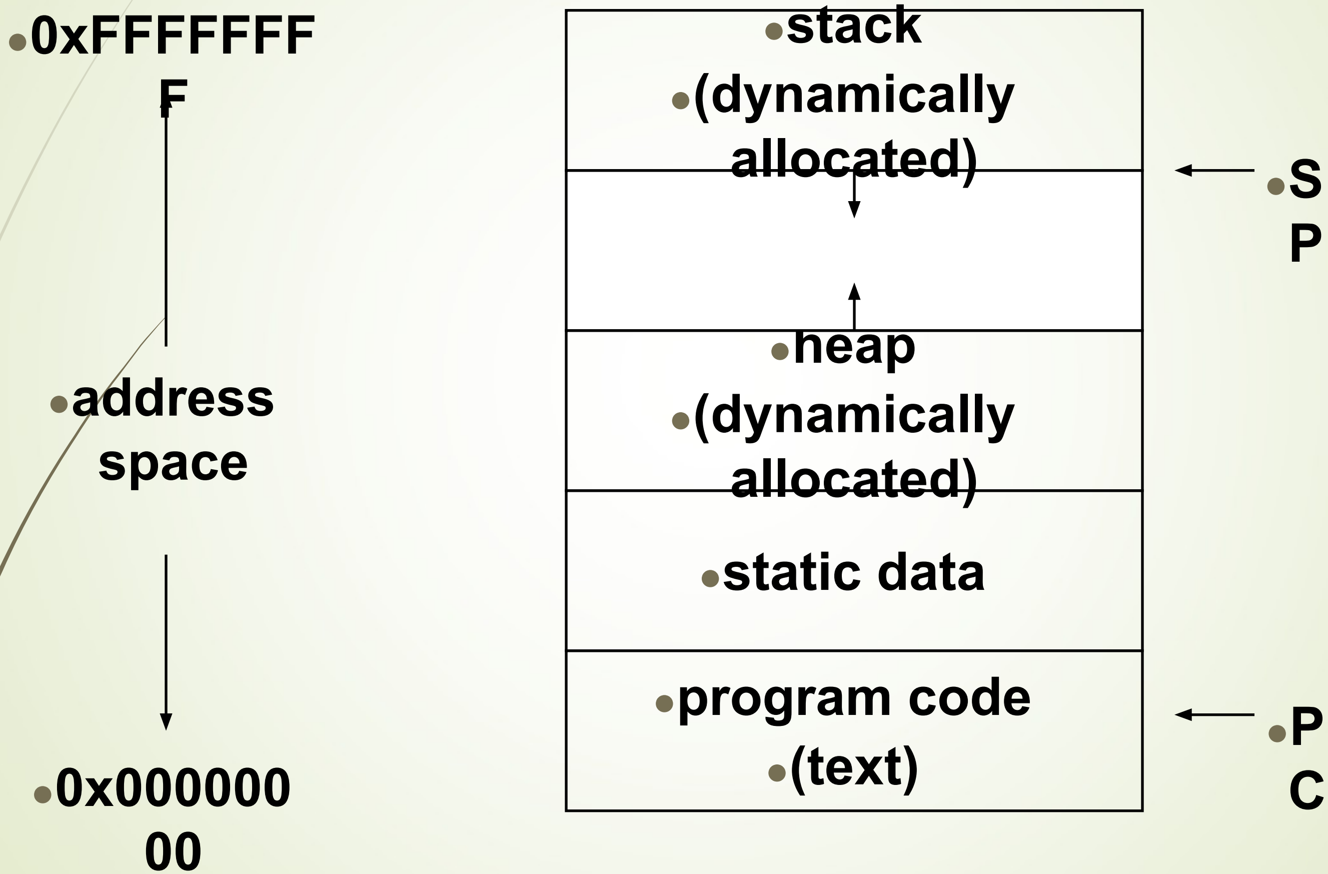
Eight Queens

Place eight queens on the chessboard such that no queen attacks any other one.

queens[0]	0
queens[1]	4
queens[2]	7
queens[3]	5
queens[4]	2
queens[5]	6
queens[6]	1
queens[7]	3



Typical Memory for Running Program
(Windows & Linux)



Another Example

Traversing through a directory or file system
Traversing through a tree of search results.

Tail recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.

Mutual Recursion

- Mutual Recursion: Functions calling each other.
- Let's say FunA calling FunB and FunB calling FunA recursively.
- This is not actually not recursive but it's doing same as recursive.
- So you can say Programming languages which are not supporting recursive calls, mutual recursion can be applied there to fulfill the requirement of recursion.
- Base condition can be applied to any into one or more than one or all functions.

Home Work

- Read about Nested Recursion gives one example of Nested Recursion
- Read about Palindrome problem and solve it through Recursion